Information and Computation **158**, 18–28 (2000)
doi:10.1006/inco.1999.2814, available online at http://www.idealibrary.com on IDEAL®

Parallel Preprocessing for Path Queries

Torben Hagerup

Fachbereich Informatik, Johann Wolfgang Goethe-Universität Frankfurt, Robert-Maier-Strasse 11–15, D-60054 Frankfurt am Main, Germany
E-mail: hagerup@informatik.uni-frankfurt.de

We consider the problem of preprocessing a tree T with edge labels drawn from a semigroup such that subsequent queries for the semigroup product of the edge labels on a path in T can be answered efficiently. A sequential algorithm exhibiting an optimal trade-off between preprocessing time and query time was described by Chazelle. A parallelization of the preprocessing part of Chazelle's algorithm for the exclusive-read exclusive-write parallel RAM (EREW PRAM) was announced by Alon and Schieber, but few details were provided. Later a different solution, complete with all details, was described by Thorup, but it requires the stronger concurrent-read exclusive-write PRAM. We describe a simple algorithm for the EREW PRAM.

1. INTRODUCTION

A *semigroup* is a pair $(S, \circ)$, where S is a set and $\circ$ is an associative operation on S. The *path-query problem* with respect to a semigroup $(S, \circ)$ for a directed or undirected graph G, each of whose edges e is labeled with an element $\ell(e)$ of S, consists in preprocessing G to enable subsequent *path queries* to be answered quickly. A path query specifies two vertices u and v in G with the property that G contains a unique (simple) path P from u to v, and the required answer is the semigroup product $\ell(e_1) \circ \cdots \circ \ell(e_k)$, where $e_1, ..., e_k$, in that order, are the edges on P. A semigroup $(S, \circ)$ is *efficient* if $a \circ b$ can be computed from a and b in constant time by a single processor for all $a, b \in S$. All semigroups considered in the following are assumed to be efficient.

The path-query problem was studied by Yao [16] for the case in which G is a directed path (then the problem is more appropriately called the *range-query problem*). Forgoing the algorithmic details, he provided matching upper and lower bounds in the form of a tight trade-off between preprocessing time and query time: For n-vertex paths, if we spend $O(m)$ preprocessing time, where $m \geqslant n$, the

[1] Part of the work was carried out while the author was with the Max-Planck-Institut für Informatik in Saarbrücken, Germany.

0890-5401/00 $35.00

achievable query time is $\Theta(\alpha(m, n))$, where α is Tarjan's "inverse-Ackermann" function, defined in the following section. Yao's result was generalized from directed paths to undirected trees without any detriment to the resource bounds by Chazelle [5], who also gave the details of a RAM implementation.

The possibility of a parallelization of the path-query preprocessing was first discussed by Alon and Schieber [3]. (Because queries are answered extremely fast even sequentially, parallelizing the query algorithm appears rather pointless and has not been attempted.) Alon and Schieber claimed (in slightly different terms) that for all $m \geqslant n$, the preprocessing of an n-vertex undirected tree for answering subsequent path queries in $O(\alpha(m, n))$ time can be carried out in $O(\log n)$ time on an exclusive-read exclusive-write parallel RAM (EREW PRAM) with $O(m/\log n)$ processors. The time bound of $O(\log n)$ is optimal for the given model of computation, and the number of *operations* executed, the product of the running time and the number of processors employed, is $O(m)$, i.e., within a constant factor of the running time of the sequential algorithm, which means that the parallel algorithm has *optimal speedup*. Alon and Schieber did not provide any details of their parallel algorithm, however, except noting that it is similar to the sequential algorithm and based on standard techniques of parallel computation. Thorup [15] later cast doubt on the existence of a parallel algorithm of the kind outlined by Alon and Schieber and provided new sequential and parallel algorithms. Although Thorup's sequential algorithm may be somewhat simpler than that of Chazelle, we contend that Thorup's exposition is still involved and hard to follow. More importantly, he claims his parallel result only for the stronger concurrent-read exclusive-write (CREW) PRAM, leaving the status of the corresponding question for the EREW PRAM somewhat in the lurch. We resolve the matter conclusively and vindicate Alon and Schieber by showing that the result for the EREW PRAM can be obtained with little effort by appealing, for the most part, to standard parallel techniques, namely prefix summation, tree contraction, and the Euler-tour technique. The work reported here was motivated by a desire to provide a firm basis for an application of the EREW PRAM result of Alon and Schieber to the solution of problems on graphs of bounded treewidth [7].

The paradigm underlying our solution that sets it off most clearly from that of Thorup and is the key to its simplicity ranks among those most heavily employed in the design of parallel algorithms: In order to solve a given problem efficiently in parallel, divide it into suitable subproblems, solve each subproblem using an efficient sequential algorithm (assumed to exist), with all subproblems being solved in parallel, and finally obtain the overall solution by combining the solutions of the subproblems as appropriate, probably using a not-so-efficient parallel algorithm. It is easy to understand the popularity of this strategy, which we call *sequentializing divide-and-conquer*. In the context of paper design and exposition, it saves effort because all the intricacy of the sequential algorithm is wrapped up nicely in a "black box" and need not be reconsidered; in particular, this will be a great relief here. And in the context of concrete parallel implementations, the fact that each processor spends much of its time executing a sequential algorithm means that little time is lost in communication with other processors. We illustrate the principle through a simple example: Suppose that the task is to compute the sum of n

numbers, where n is a power of two. The most natural parallel algorithm views the n numbers as the leaves of a complete binary tree whose internal vertices are $n-1$ processors. By sending values up the tree towards the root and letting each processor add the values that it receives from its children before sending the sum to its parent, we can obviously sum the n numbers in $O(\log n)$ time. Alternatively, and using sequentializing divide-and-conquer, we can divide the n numbers into $O(n/\log n)$ groups of $O(\log n)$ numbers each, sum the numbers within each group sequentially, and sum only the group sums using the binary tree. We achieve the same running time as before (up to a constant factor), but now using only $O(n/\log n)$ processors and, therefore, $O(n)$ operations. Although the tree-summation algorithm in itself does not have optimal speedup, the complete algorithm does. Our parallel algorithm with optimal speedup for the path-query preprocessing will be derived from a nonoptimal parallel algorithm in much the same way. In the following, when speaking of a *near-optimal* algorithm for path-query preprocessing, we mean an algorithm for the EREW PRAM that processes n-vertex graphs for subsequent constant-time queries using $O(\log n)$ time and $O(n \log n)$ operations—informally speaking, such an algorithm misses optimal speedup by (almost) a logarithmic factor.

In the context of the path-query problem for trees, the division into subproblems, although not difficult, is not trivial, but suitable parallel algorithms are already available. We can use Chazelle's algorithm for the sequential part. The combination of the subproblem solutions is essentially a smaller instance of the path-query problem, and it turns out to be easy to design a near-optimal parallel algorithm that can be used here.

We assume that each processor is able to execute the operations of addition, subtraction, and left and right bit shifts by a variable number of bit positions on $O(\log n)$-bit integers in constant time. Formulating the problem of computing the lowest common ancestor (LCA) of two vertices in a tree as a special case of a path query, we obtain a preprocessing algorithm that allows subsequent LCA queries to be executed in constant time and that preprocesses an n-vertex tree in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors using no arithmetic operations besides those listed above. To our knowledge, no algorithm with these characteristics was formulated previously. E.g., an algorithm of Schieber and Vishkin [12] needs several additional operations, namely at least bitwise exclusive-or and "base-two discrete logarithm," i.e., the operation $x \mapsto \lfloor \log_2 x \rfloor$. According to a remark in [12, p. 1259], the algorithm needs even multiplication and division.

In Section 3 we describe a near-optimal preprocessing algorithm for path queries on directed paths. The algorithm for directed paths is extended to directed, undirected, and so-called bidirected trees in Section 4, and in Section 5 we complete the description of how to derive a parallel algorithm with optimal speedup for the problem at hand from the near-optimal one developed in Section 4.

2. PRELIMINARIES

We define the function α following [13]. First, for all nonnegative integers i and j, $A(i, j)$ is defined by

$$A(i, j) = \begin{cases} 2j, & \text{if } i = 0 \text{ or } j \leqslant 1; \\ A(i-1, A(i, j-1)), & \text{if } i \geqslant 1 \text{ and } j \geqslant 2. \end{cases}$$

The function A is nondecreasing in each argument, and $\lim_{i \to \infty} A(i, 4) = \infty$. Now, for all positive integers m and n, $\alpha(m, n) = \min\{i \geqslant 1 \mid A(i, 4\lceil m/n \rceil) > \log_2 n\}$.

We assume a representation of graphs in which every vertex has an adjacency list with an entry for each of its incident edges, and the two entries of each edge in the adjacency lists of its endpoints are linked via *cross pointers*. By an *S-labeled graph*, where S is a set, we mean a directed or undirected graph, each of whose edges is labeled with an element of S. For every semigroup $(S, \circ)$, the $\circ$-*value* of a path P in an S-labeled graph is $a_1 \circ \cdots \circ a_k$, where $a_1, ..., a_k$, in that order, are the labels of the edges on P. When $\circ$ is an associative operation on a set S, we denote by $\circ_{\mathbf{R}}$ the associative operation on S with $a \circ_{\mathbf{R}} b = b \circ a$ for all $a, b \in S$. For every semigroup $(S, \circ)$, we will assume without loss of generality the existence of a *neutral element* ε with $a \circ \varepsilon = \varepsilon \circ a = a$ for all $a \in S$. The *depth* of a vertex v in a rooted tree T, denoted depth(v), is the number of edges on the path from v to the root of T. An *intree* is a rooted tree, each of whose edges is directed towards the root.

3. PATH QUERIES ON DIRECTED PATHS

In this section we describe a near-optimal preprocessing algorithm for path queries on directed paths. The description is designed to facilitate the extension to the case of trees carried out in the following section. The lemmas below deal with the computation of the values of certain paths in intrees.

LEMMA 1. *For all efficient semigroups $(S, \circ)$ and all integers $n \geqslant 2$, the following problem can be solved on an EREW PRAM using $O(\log n)$ time and $O(n)$ operations: Given a collection of vertex-disjoint S-labeled intrees of altogether n vertices, mark each vertex v in such a tree T with the $\circ$-value of the path in T from v to the root of T.*

Proof. If there is only one tree, the claim follows by a standard application of tree contraction [1, Theorem 3.2]. The case of several trees can be reduced to the case of one tree by introducing a new vertex r and an edge labeled ε from each tree root to r. ∎

If Lemma 1 is thought of as solving a *values-to-root problem*, Lemma 2 below solves a corresponding *values-from-leaves problem* with respect to a semigroup $(S, \circ)$. Formally, we are again given a collection of vertex-disjoint S-labeled intrees, and the task is to mark each leaf v in such a tree T with an array $C_v[0..d]$, where d is the depth of v in T, such that for $i = 0, ..., d$, $C_v[i]$ is the $\circ$-value of the path in T from v to that ancestor of v whose distance from v is i. We define the *output size* of an instance of the values-from-leaves problem given by a collection of trees with the leaf set V as the total size of the arrays to be computed, i.e., as $\sum_{v \in V} (\text{depth}(v) + 1)$.

LEMMA 2. *For all integers $m \geqslant 2$, instances of output size m of the values-from-leaves problem with respect to an efficient semigroup $(S, \circ)$ can be solved on an EREW PRAM using $O(\log m)$ time and $O(m)$ operations.*

Proof. The case of several trees again reduces trivially to the case of a single tree, so we will assume the instance at hand, of output size m, to be given by a single tree T. We begin by constructing a directed graph G that is the disjoint union of all leaf-to-root paths in T, with edge labels in G derived in the obvious way from those of T. After the construction of G, described below, we form a new graph $G_{\mathbf{R}}$ from G by replacing each edge (u, v) in G by the reverse edge (v, u), with the same label as (u, v). Considering each path in $G_{\mathbf{R}}$ as a degenerate tree, we then use the algorithm of Lemma 1 to compute the $\circ_{\mathbf{R}}$-value of each path in $G_{\mathbf{R}}$ from a vertex u to its root, the root v of the tree containing u, which is the same as the $\circ$-value of the path in T from v to u. It is easy to see that this computes all required $\circ$-values. These values can be stored in appropriate arrays by repeated application of the algorithm of Lemma 1 (for several fixed semigroups): First each vertex v in G is informed of its depth in T, after which appropriately sized arrays C_v can be allocated to the leaves v of T by means of a prefix summation (described, e.g., in [9, Section 2.1]). Subsequently, for each head v of a path P in $G_{\mathbf{R}}$, the address of (the first cell of) the array C_v is broadcast to all vertices on P, after which the $\circ$-values computed above of paths in T starting at v can be stored in C_v in constant time.

In order to construct the graph G, we begin by ordering the children of each vertex in T arbitrarily from left to right and using the Euler-tour technique [14] in connection with optimal-speedup list ranking (which can be viewed as solving a special case of the values-to-root problem) to number the leaves in T consecutively from left to right and to mark each vertex v in T with the smallest and largest numbers of a leaf descendant of v. Each vertex v then computes $W(v)$, defined as the number of leaf descendants of v, and a prefix summation is used to allocate to each vertex v in T an array of $W(v)$ new vertices, the ith of which, for $i = 1, ..., W(v)$, will be said to *correspond* to the ith leaf descendant of v, counted from the left. These new vertices are the vertices of G. In order to obtain the edges of G, we connect all pairs of new vertices in the arrays of adjacent vertices in T that correspond to the same leaf, which is easy to do if each vertex in T first communicates to each of its children the address of its block of new vertices and the smallest and largest numbers of its leaf descendants (again, a special case of the values-to-root problem). Since G contains exactly m vertices, both the construction of G and the subsequent processing can be carried out in $O(\log m)$ time using $O(m)$ operations. ▌

We now return to the path-query problem with respect to an efficient semigroup $(S, \circ)$. Consider an instance of the problem given by an n-vertex directed S-labeled path P, which we again consider to be a degenerate intree $T = (V, E)$. We begin by using the algorithm of Lemma 1 to mark each vertex in T with its depth. Take $h = \lceil \log_2 n \rceil$ and $N = 2^h$. For reasons that will not become clear until the next section, we introduce an integer parameter $q \in \{1, ..., N\}$ called the *offset*; in the context of the present section, any choice of q such as $q = 1$ will do. We define the *q-level* of a nonnegative integer k, denoted $L_q(k)$, as the largest nonnegative integer d such that 2^d divides $q + k$. E.g., $L_1(2) = 0$ and $L_1(3) = 2$.

For $d = 1, ..., h$, we create a copy of T and carry out the following preprocessing, all values of d being treated in parallel: First we remove from (the relevant copy of)

T every edge leaving a vertex whose depth has q-level d or more, in effect turning each such vertex into the root of a new tree. Then the algorithm of Lemma 1 is used to solve the resulting instance of the values-to-root problem, for each vertex v storing the $\circ$-value of the path from v to its root in $B_v[d]$ for a suitable array B_v. Subsequently we reinsert the edges removed above, instead remove every edge entering a vertex whose depth has q-level d or more, and use the algorithm of Lemma 2 to solve the resulting instance of the values-from-leaves problem, storing the values computed for each vertex v in an array C_v. This essentially ends the preprocessing, which can be carried out in $O(\log n)$ time using $O(hn) = O(n \log n)$ operations.

Suppose that a query asks for the $\circ$-value of the path in T from v to u, where u is a proper ancestor of v. Take $i = \text{depth}(u)$ and $j = \text{depth}(v)$. If d is the q-level of the unique integer k with $i \leqslant k \leqslant j$ of maximum q-level, $B_v[d]$ indicates the $\circ$-value of the path in T from v to the ancestor w of v of depth k, and the $\circ$-value of the path from w to u can be found in $C_w[k-i]$, so that the query can be answered in constant time once k, d, and w are known. We describe how to compute these quantities.

Consider the representations of $q+i$, $q+j$ and $q+k$ as binary strings of $h+1$ bits each. It can be seen that if the binary representations of $q+i$ and $q+j$ have the string β as their longest common prefix, i.e., are of the form $\beta 0\gamma$ and $\beta 1\gamma'$, then the binary representation of $q+k$ is $\beta 0\gamma$ if $\gamma = 0 \cdots 0$, and otherwise is $\beta 100 \cdots 0$. It follows that k can be computed in constant time once $|\beta| = h - \lfloor \log_2((q+i) \oplus (q+j)) \rfloor$ is known, where $\oplus$ denotes bitwise exclusive-or of $(h+1)$-bit strings. The operations $\oplus$ and $x \to \lfloor \log_2 x \rfloor$ are not part of our instruction repertoire, but can be executed in constant time during a query by lookup in tables of size $O(n)$ constructed as part of the preprocessing. Since each entry in the relevant tables can certainly be computed in $O(\log n)$ time by a single processor, this is obvious in the case of the unary operation $x \to \lfloor \log_2 x \rfloor$. At first glance, the binary operation $\oplus$ might appear to need a table of size $\Theta(n^2)$, whose construction would be too expensive. However, since we can compute $(q+i) \oplus (q+j)$ by breaking the bit strings $q+i$ and $q+j$ into halves and applying the $\oplus$ operation separately to the first halves and to the last halves, it suffices to provide a table of size $O(n)$, which is readily constructed. Similarly, d can be computed from k using table lookup. In order to determine w, finally, we extend the computation of $B_v[d]$ so that it computes not only the $\circ$-value of the path from v to w, but also w itself. This can be done by replacing the semigroup $(S, \circ)$ by the semigroup $(S \times V, \circ')$, where $(a, x) \circ' (b, y) = (a \circ b, y)$ for all $a, b \in S$ and all $x, y \in V$.

Note that table lookup is needed only during the processing of queries and not during the preprocessing. This is crucial because our model does not allow concurrent reading: The processors could not share a common set of tables and would need private tables, the construction of which would be too expensive.

We summarize the result of this section as follows.

LEMMA 3. *There is a near-optimal preprocessing algorithm for path queries on directed paths with respect to efficient semigroups.*

4. PATH QUERIES IN TREES

In this section we generalize the construction of the previous section from directed paths to intrees and subsequently to general trees. We develop further a technique of Thorup [15] of dividing a tree into suitable pieces by cutting it at all vertices of regularly spaced depths.

The description given near the end of the previous section of preprocessing and query algorithms for the case of directed paths actually never made any use of the fact that the input tree T was degenerate, and we will use precisely the same algorithms in the case of a general intree $T=(V, E)$. Only the resource requirements of the preprocessing algorithm need reconsideration, the reason being that, whereas the output size is within a constant factor of the input size when the algorithm of Lemma 2 is applied to a path, this is not true in the case of a general intree. This is where the offset q becomes important. Indeed, the total output size of all applications of the algorithm of Lemma 2 is bounded by

$$\sum_{v\in V}\sum_{i=1}^{L_q(\mathrm{depth}(v))} 2^i \leqslant 2\sum_{v\in V} 2^{L_q(\mathrm{depth}(v))}.$$

Let us define $M(q)$ as the right-hand side above, i.e., $M(q)=2\sum_{v\in V} 2^{L_q(\mathrm{depth}(v))}$. Since the preprocessing can be executed in $O(\log n)$ time using $O(n\log n+M(q))$ operations, our remaining task is to demonstrate that there is an offset $q\in\{1, ..., N\}$ with $M(q)=O(n\log n)$ and that such an offset can be found efficiently.

Observe first that for all integers $t\geqslant 1$, $\sum_{i=0}^{t-1} 2^{L_1(i)}\leqslant t(\log_2 t+1)$ (e.g., estimate the contribution of the terms of each possible magnitude separately). The existence of a q with $M(q)=O(n\log n)$ now follows by writing

$$\sum_{q=1}^{N} M(q)=2\sum_{q=1}^{N}\sum_{v\in V} 2^{L_q(\mathrm{depth}(v))}=2\sum_{v\in V}\sum_{q=1}^{N} 2^{L_1(\mathrm{depth}(v)+q-1)}$$
$$\leqslant 2\sum_{v\in V}\sum_{i=0}^{2N-1} 2^{L_1(i)}\leqslant 4Nn(h+2)$$

and noting that not all terms in the left-hand sum can be larger than their average, which is bounded by $4n(h+2)=O(n\log n)$.

We finally show how to compute $M(1), ..., M(N)$, after which it is a simple matter to choose the offset q to minimize $M(q)$ and to carry out the preprocessing as described above.

Given a sequence $(n_0, ..., n_{N-1})$ of N integers and a sequence $(c_1, c_2, ...)$ of at least $2N-1$ integers, we define $(n_0, ..., n_{N-1})\otimes(c_1, c_2, ...)$ to be the sequence $(m_1, ..., m_N)$ with $m_q=\sum_{i=0}^{N-1} n_i c_{q+i}$ for $q=1, ..., N$. Then

$$(M(1), ..., M(N))=(n_0, ..., n_{N-1})\otimes(c_1, c_2, ...),$$

where n_i is twice the number of vertices of depth i in the input tree T for $i=0, ..., N-1$ and $(c_1, c_2, ...)=(1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, ...)$; i.e., $c_i=2^{L_1(i-1)}$ for

$i = 1, 2, \ldots$ We first compute $(n_0, \ldots, n_{N-1})$ by sorting the vertices in T according to their depths [2, 6] or, alternatively, using the procedure of [10]. Now

$$\begin{aligned}
&(n_0, \ldots, n_{N-1}) \otimes (1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, \ldots)\\
=&(n_0, \ldots, n_{N-1}) \otimes (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \ldots)\\
+&(n_0, \ldots, n_{N-1}) \otimes (0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots)\\
+&(n_0, \ldots, n_{N-1}) \otimes (0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, \ldots)\\
+&(n_0, \ldots, n_{N-1}) \otimes (0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, \ldots)\\
+&\cdots,
\end{aligned}$$

with altogether $h+1$ terms (added componentwise, of course), and each term is easy to compute in $O(\log n)$ time using $O(n)$ operations. E.g., in order to find $(n_0, \ldots, n_{N-1}) \otimes (0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, \ldots)$, we compute $s_i = 2(n_i + n_{4+i} + n_{8+i} + \cdots)$ for $i = 0, \ldots, 3$ and output $(s_3, s_2, s_1, s_0, s_3, s_2, s_1, s_0, s_3, \ldots)$. This shows that $M(1), \ldots, M(N)$ can be determined in $O(\log n)$ time using $O(n \log n)$ operations. We have proved:

LEMMA 4. *There is a near-optimal preprocessing algorithm for path queries on intrees with respect to efficient semigroups.*

We finally take the step from intrees, in which every path goes from a vertex to one of its ancestors, to undirected trees, where this is no longer true. Consider a semigroup $(S, \circ)$ and a rooted undirected S-labeled input tree T and let u and v be vertices in T. The $\circ$-value of the path from u to v can be obtained by composing the $\circ$-value of the path from u to the lowest common ancestor (LCA) w of u and v with the $\circ_{\mathbf{R}}$-value of the path from v to w. Thus, a query in T reduces to two queries in intrees with as many vertices as T plus the determination of the depth of the LCA of two vertices in T (our query algorithm in Section 3 does not need to know the LCA itself). As observed by Berkman and Vishkin [4], the latter problem easily reduces to the path-query problem for a directed path equal to an Euler tour of T in which each vertex is represented by its depth in T and the path is preprocessed for range-minimum queries: Each vertex in T is equipped with a pointer to one of its occurrences on the Euler tour, and the depth of the LCA of two vertices u and v is simply the minimum value occurring between the pointers of u and v (inclusive).

As follows easily from the preceding argument, we can actually allow each edge in T to be associated with two elements of S, one to be used with each direction of traversal. Put differently, we can extend the result from undirected trees to *bidirected trees*, where we define a bidirected tree to be a directed graph whose underlying undirected graph is a tree and that, with each edge (u, v), contains also the reverse edge (v, u).

LEMMA 5. *There is a near-optimal preprocessing algorithm for path queries on bidirected trees with respect to efficient semigroups.*

5. ACHIEVING OPTIMAL SPEEDUP

In this section we go from the near-optimal preprocessing algorithm of the previous section to one with optimal speedup.

Let $(S, \circ)$ be a semigroup, let $T=(V, E)$ be a rooted bidirected S-labeled input tree with $n \geqslant 3$ vertices, and suppose that our task is to preprocess T using $O(m)$ operations, where $m \geqslant n$, to allow subsequent queries for $\circ$-values of paths in T to be answered in $O(\alpha(m, n))$ time. If $m=\Omega(n \log n)$, the near-optimal algorithm of the previous section can be used directly, so assume that $m=O(n \log n)$. We will also assume T to be binary, which can easily be achieved by replacing each vertex in T with $d \geqslant 3$ children by a binary tree with d leaves, all of whose edges are labeled with ε. We choose n' as an integer with $3 \leqslant n' \leqslant n$ and $n'=\Theta(n \log n/m)$ and take $m'=\lceil mn'/n \rceil$. Informally, the reasoning behind this choice of parameters is that every subtree of T with at most n' vertices can be processed in $O(m')=O(\log n)$ time for a query time of $O(\alpha(m', n'))=O(\alpha(m, n))$, where the latter relation follows from the inequalities $n' \leqslant n$ and $m'/n' \geqslant m/n$. Correspondingly, we essentially divide T into pieces of size n'. More precisely, we color the edges in E such that each edge has the same color as its reverse edge, calling each subgraph of T spanned by a maximal set of edges of the same color a *piece* and calling each vertex in T shared between two or three pieces a *boundary vertex*, such that each piece is a (bidirected) tree containing at most n' vertices, at most two of which are boundary vertices, and such that the total number of pieces is $O(n/n')$. It is shown in [11, Section 3.3.5] that this can be done in $O(\log n)$ time using $O(n)$ operations as a simple application of tree contraction. We now allocate a processor to each piece H (e.g., allocating the processor to the vertex in H of minimal depth by means of a prefix summation) and let this processor preprocess the piece in $O(m')$ time for query time $O(\alpha(m, n))$ using the sequential algorithm of Chazelle [5]. The total number of operations needed for this is $O(m'n/n')=O(m)$. Subsequently we create a bidirected tree T' on the set of boundary vertices with, for each piece that contains two boundary vertices u and v, edges (u, v) and (v, u) labeled with the $\circ$-values of the path from u to v and from v to u, respectively, in the original tree T. T' has $O(n/n')$ vertices and is preprocessed for path queries using the near-optimal algorithm of Lemma 5. This also needs $O(\log n)$ time and $O(n \log n/n')=O(m)$ operations and ends the preprocessing.

In order to answer a query for the $\circ$-value of the path in T from u to v, where $u, v \in V$, we proceed as follows: Let H_u and H_v be pieces containing u and v, respectively. If $H_u=H_v$, the query is answered via a query to the data structure computed for H_u. Otherwise, let r_u be the boundary vertex of H_u closest to v, and let r_v be the boundary vertex of H_v closest to u; we describe below how to determine r_u and r_v. Now the $\circ$-value of the path from u to v can be found by composing the $\circ$-values of the paths in T from u to r_u, from r_u to r_v, and from r_v to v, which in turn are obtained via queries to the data structures computed for H_u, T', and H_v, respectively. In order to find r_u and r_v, we observe that this can be done by trying out all the pairs (at most four pairs) of boundary vertices if $S=(\mathbb{N}, +)$ and each edge is labeled with the integer 1: (r_u, r_v) is simply the pair whose distance is minimal. It therefore suffices to carry out the preprocessing not just for the semigroup

$(S, \circ)$ under consideration, but additionally for the semigroup $(\mathbb{N}, +)$. The query time is clearly $O(\alpha(m, n))$.

THEOREM 6. *For all efficient semigroups* $(S, \circ)$ *and for all integers n and m with* $m \geqslant n \geqslant 2$, *every n-vertex bidirected S-labeled tree T can be preprocessed on an EREW PRAM using* $O(\log n)$ *time and* $O(m)$ *operations such that subsequent queries for the* $\circ$*-value of a path in T can be answered in* $O(\alpha(m, n))$ *time by a single processor.*

For each semigroup for which a more efficient sequential preprocessing is possible, we can derive a more efficient parallel preprocessing algorithm. An important such case is the computation of least common ancestors of pairs of vertices in a tree $T = (V, E)$, which can be formulated as a path-query problem by labeling each vertex v with $(v, \text{depth}(v))$ and considering the semigroup $(V \times (\mathbb{N} \cup \{0\}), \circ)$ with

$$(u, i) \circ (v, j) = \begin{cases} (u, i), & \text{if } i \leqslant j; \\ (v, j), & \text{if } i > j. \end{cases}$$

A linear-time sequential preprocessing algorithm for constant-time LCA queries was described by Harel and Tarjan [8]. We can exploit this in the parallel algorithm given above by changing the piece size n' to $\Theta(\log n)$, which yields a preprocessing algorithm for the EREW PRAM that uses $O(\log n)$ time and $O(n)$ operations and allows subsequent LCA queries to be answered in constant time. As discussed in the Introduction, a similar result was described previously by Schieber and Vishkin [12], but our algorithm uses a more restricted instruction set.

Received March 27, 1998; final manuscript received July 2, 1999

REFERENCES

1. Abrahamson, K., Dadoun, N., Kirkpatrick, D. G., and Przytycka, T. (1989), A simple parallel tree contraction algorithm, *J. Algorithms* **10**, 287–302.
2. Ajtai, M., Komlós, J., and Szemerédi, E. (1983), An $O(n \log n)$ sorting network, *in* "Proceedings, 15th Annual ACM Symposium on Theory of Computing," pp. 1–9.
3. Alon, N., and Schieber, B. (1987), "Optimal Preprocessing for Answering On-Line Product Queries," Tech. Rep. No. 71/87, Tel Aviv University.
4. Berkman, O., and Vishkin, U. (1993), Recursive star-tree parallel data structure, *SIAM J. Comput.* **22**, 221–242.
5. Chazelle, B. (1987), Computing on a free tree via complexity-preserving mappings, *Algorithmica* **2**, 337–361.
6. Cole, R. (1988), Parallel merge sort, *SIAM J. Comput.* **17**, 770–785.
7. Hagerup, T. (1997), Dynamic algorithms for graphs of bounded treewidth, *in* "Proceedings, 24th International Colloquium on Automata, Languages and Programming," Lecture Notes in Computer Science, Vol. 1256, pp. 292–302, Springer-Verlag, Berlin.
8. Harel, D., and Tarjan, R. E. (1984), Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.* **13**, 338–355.
9. JáJá, J. (1992), "An Introduction to Parallel Algorithms," Addison–Wesley, Reading, MA.

10. Katajainen, J., Thorup, M., and Träff, J. L. (1995), "Parallel Tree Slicing," Tech. Rep. No. 95/13, Dept. of Computer Science, Univ. of Copenhagen.
11. Reid-Miller, M., Miller, G. L., and Modugno, F. (1993), List ranking and parallel tree contraction, *in* "Synthesis of Parallel Algorithms" (J. H. Reif, Ed.), Chap. 3, pp. 115–194, Morgan Kaufmann, San Mateo, CA.
12. Schieber, B., and Vishkin, U. (1988), On finding lowest common ancestors: Simplification and parallelization, *SIAM J. Comput.* **17**, 1253–1262.
13. Tarjan, R. E. (1975), Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22**, 215–225.
14. Tarjan, R. E., and Vishkin, U. (1985), An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14**, 862–874.
15. Thorup, M. (1997), Parallel shortcutting of rooted trees, *J. Algorithms* **23**, 139–159.
16. Yao, A. C. (1982), Space-time tradeoff for answering range queries, *in* "Proceedings, 14th Annual ACM Symposium on Theory of Computing," pp. 128–136.